

Name:Devarsh Patel

Nihal Kumar

Dhruv Mittal

Reg.No:17BCE0874

17BCE0632

17BCE2110

Faculty:Dr.Padma Priya R

- : Review - 99 : -

Data Structure and Algorithm

Bin Packing Algorithm

Abstract:

Bin packing problem with conflicts(BPPC) is a complex combinatorial optimization problem originated from logistics, whose objective is packing all the items with the least number of bins and satisfying the conflict constraints among the items. In this paper, we firstly give the description and 0-1 integer programming model of BPPC, and then transform the model into the representation of conflict graph structure. After that, an ant colony optimization(ACO) algorithm framework is proposed for solving the conflict elimination procedure of BPPC based on a graph coloring heuristic, according to the results of conflict elimination, an improved first-fit decreasing algorithm is used to finish the packing operations of the subsets of items without conflicts. The experiments show that the improve ACO algorithm in this paper is valid and could provide a feasible and high-quality solutions of BPPC efficiently.

Introduction:

In the **bin packing problem**, objects of different volumes must be packed into a finite number of bins or containers each of volume V in a way that minimizes the number of bins used.

In computational complexity theory, it is a combinatorial NP-hard problem.[1] The decision problem (deciding if objects will fit into a specified number of bins) is NP-complete.[2]

There are many variations of this problem, such as 2D packing, linear packing, packing by weight, packing by cost, and so on. They have many applications, such as filling up containers, loading trucks with weight capacity constraints, creating file backups in media and technology mapping in Field-programmable gate array semiconductor chip design. The bin packing problem can also be seen as a special case of the cutting stock problem. When the number of bins is restricted to 1 and each item is characterised by both a volume and a value, the problem of maximising the value of items that can fit in the bin is known as the knapsack problem.

A variant of bin packing that occurs in practice is when items can share space when packed into a bin. Specifically, a set of items could occupy less space when packed together than the sum of their individual sizes. This variant is known as VM packing[4] since when **Virtual machines (VMs)** are packed in a server, their total memory requirement could decrease due to pages shared by the VMs that need only be stored once. If items can share space in arbitrary ways, the bin packing problem is hard to even approximate. However, if the space

sharing fits into a hierarchy, as is the case with memory sharing in virtual machines, the bin packing problem can be efficiently approximated. Another variant of bin packing of interest in practice is the so-called online bin packing. Here the objects of different volume are supposed to arrive sequentially and the decision maker has to decide whether to select and pack the currently observed item, or else to let it pass.

Types of algorithm for Bin packing:

- 1)First Fit
- 2)Best Fit
- 3)Offline Algorithm
- 4)First fit decreasing
- 5)Next fit

Bin Packing can be Done in ***Classical Approach(Basic)*** as well as ***Heuristic Approach***.

1)Classical Approach(Basic):

In classical Approach we generally compares the size of the elements with the Bin size and by comparing we just add the weight to the Bin.

E.g:

This is a C Program to implement Bin packing algorithm. This is a sample program to illustrate the Bin-Packing algorithm using next fit heuristics. In the bin packing problem, objects of different volumes must be packed into a finite number of bins or containers each of volume V in a way that minimizes the

number of bins used. In computational complexity theory, it is a combinatorial NP-hard problem.

There are many variations of this problem, such as 2D packing, linear packing, packing by weight, packing by cost, and so on. They have many applications, such as filling up containers, loading trucks with weight capacity constraints, creating file backups in media and technology mapping in Field-programmable gate array semiconductor chip design.

Here is source code of the C Program to Implement the Bin Packing Algorithm:

```
#include<stdio.h>

void binPacking(int *a, int size, int n) {
    int binCount = 1, i;
    int s = size;
    for (i = 0; i < n; i++) {
        if (s - *(a + i) > 0) {
            s -= *(a + i);
            continue;
        } else {
            binCount++;
            s = size;
            i--;
        }
    }
}

printf("Number of bins required: %d", binCount);
}

int main(int argc, char **argv) {
    printf("Enter the number of items in Set: ");
    int n;
    int a[n], i;
```

```

    int size;
    scanf("%d", &n);
    printf("Enter %d items:", n);
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
    printf("Enter the bin size: ");
    scanf("%d", &size);
    binPacking(a, size, n);
    return 0;
}

```

Valid for Linux!!!!!!

\$ gcc bin_packing.c

\$./a.out

Output:

```

Enter the number of items in Set: 5
Enter 5 items:12 23 34 45 56
Enter the bin size: 70
Number of bins required: 3

```

So the above method gives the classical approach of Bin packing algorithm.

2)Heuristic Approach:

Several combinatorial optimization problems can be formulated as large set-covering problems. In this work, we use

the set-covering formulation to obtain a general heuristic algorithm for this type of problem, and describe our implementation of the algorithm for solving two variants of the well-known (one-dimensional) bin-packing problem: the two-constraint bin-packing problem and the basic version of the two-dimensional bin-packing problem, where the objects cannot be rotated and no additional requirements are imposed. In our approach, both the “column-generation” and the “column-optimization” phases are heuristically performed. In particular, in the first phase, we do not generate the entire set of columns, but only a small subset of it, by using greedy procedures and fast constructive heuristic algorithms from the literature. In the second phase, we solve the associated set-covering instance by means of a Lagrangian-based heuristic algorithm. Extensive computational results on test instances from the literature show that, for the two considered problems, this approach is competitive, with respect to both the quality of the solution and the computing time, with the best heuristic and metaheuristic algorithms proposed so far.

2. THE PROPOSED ALGORITHMS

In this section, two proposed algorithm *A1* and *A2* are discussed. Algorithm *A1* utilizes ranging technique and classifies inputs into 4 ranges. It will be proved that this algorithm's approximation ratio is $3/2$. Furthermore, a new linear version of *FFD* algorithm is presented.

2.1. The Proposed Algorithm A1

The algorithm tries to create output bins which are at least $2/3$ full. It is proved that in this condition the approximation ratio of the algorithm is $3/2$.

As mentioned, in this algorithm inputs are classified into 4 ranges $(0-\frac{1}{3})$, $(\frac{1}{3}-\frac{1.5}{3})$, $(\frac{1.5}{3}-\frac{2}{3})$ and $(\frac{2}{3}-1)$ called S , M_1 , M_2 and L , respectively.

In first step, L items are put in separate output bins, then M_1 and M_2 are sorted. We try to match any item in M_2 with the biggest possible item in M_1 . Obviously, after that this step, some items will be remained in M_1 and M_2 . We match M_1 items with each other and add $\frac{|M_1|}{2}$ to *Bin-counter* (The number of used bins). In next step, we try to match M_2 items with S items. Finally, S items are matched with each other.

Definition1: P is the number of bins in OPT solution and P^* is the number of bins in the proposed algorithm.

Lemma1: If at least $\frac{2}{3}$ size of each output bin is full, the approximation ratio is at least $\frac{3}{2}$.

Proof: consider the worst condition that all output bins are completely full in OPT solution. Suppose that W is the sum of input items. In this condition:

$$P \geq W \text{ \& } P^* \leq \frac{W}{\frac{2}{3}} \Rightarrow P^*/P \leq \frac{3}{2} \quad \blacksquare$$

Algorithm A1:*Read inputs & classify them into S, M₁, M₂, and L**Sort M₁ & M₂***For** (any item *a* in M₂)**If** (*a* can be matched with at least an item in M₁)*Match a with the biggest possible item in M₁;**Eliminate them & Bin-counter ++;***Else** continue;*Bin-counter + = $\frac{|M_1|}{2}$;***For** (any item *a* in M₂)**Do***Choose an item b in S;**a = a + b & eliminate b & c = b;***While** (*a* ≤ 1)*Eliminate a & put c in S & Bin-counter ++;***While** (*S* is not empty)*Choose an item a in S;***While** (*a* ≤ 1)*Choose an item b in S & a = a + b & c = b;**Eliminate a & Bin-counter++ & put c in S***End**

2.2. The Proposed Algorithm A2

As mentioned, the second proposed algorithm is based on the *Firs-Fit Decreasing* algorithm. In *FFD*, the items are packed in order of non-decreasing size, and next item is always packed into the first bin in which it fits; that is, we first open bin1 and we only start bin k+1 when the current item does not fit into any of the bins 1, ..., k.

In the algorithm A2, we consider 10 classes of bins and 10 ranges of items and in any step we check at most one bin in each class. The order of choosing items and checking the bins classes are considered completely intelligently. A pseudocode of the algorithm A2 is shown.

Algorithm A2:

*Consider 10 sets of bins $B_i \forall 0 \leq i \leq 9$. A bin is in the set B_i if the bin's free space is between $(i*0.1)$ and $((i+1) 0.1)$. (At first all sets are empty;*

*Consider 10 ranges $R_i = (i * 0.1, (i + 1) * 0.1) \forall 0 \leq i \leq 9$.*

Read items from input.

Put the items into corresponding ranges

For ($i = 9; i > -1; i --$)

While (R_i is not empty)

Choose an item a in R_i randomly;

For($j = 0; j < 10; j ++$)

Choose a random bin in B_j ;

If (the bin has enough space for a)

Put a in the bin & change that to the appropriate B_i ;

Eliminate a from R_i ;

Break;

Put the item a into an empty bin and put the bin into corresponding B_i ;

Eliminate a from R_i

Obviously, the running time of the algorithm A2 is $O(n)$ (n is the number of input items) since for making decision about each item the algorithm at most spend 10 time-unit for checking 10 classes of bins.

We also can make the algorithm more efficient and consider the *Scale Parameter r* that shows the number of ranges and bins classes in the algorithm. This parameter can be chosen based on the number of inputs. For instance, if the number of inputs is 10^{10} is reasonable choose $r = 10^3$ instead of $r = 10$.

Program for first fit Decreasing:

```
#include <string.h>
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
void binPacking(int *a, int size, int n) {
```

```
    int binCount = 0, i, j;
```

```

int binValues[n];
for (i = 0; i < n; i++)
    binValues[i] = size;

for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
        if (binValues[j] - a[i] >= 0) {
            binValues[j] -= a[i];
            break;
        }
    }

for (i = 0; i < n; i++)
    if (binValues[i] != size)
        binCount++;

printf(
    "Number of bins required using first fit decreasing
algorithm is: %d",
    binCount);
}

```

```

int* sort(int *sequence, int n) {
    // Bubble Sort descending order
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n - 1; j++)
            if (sequence[j] < sequence[j + 1]) {
                sequence[j] = sequence[j] + sequence[j + 1];
                sequence[j + 1] = sequence[j] - sequence[j + 1];
            }
}

```

```

        sequence[j] = sequence[j] - sequence[j + 1];
    }

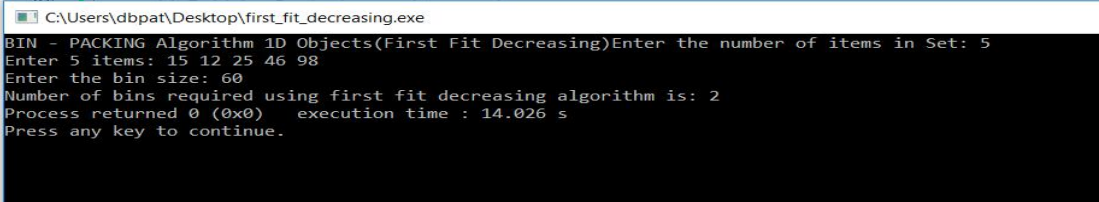
    return sequence;
}

int main(int argc, char **argv) {
    printf("BIN - PACKING Algorithm 1D Objects(First Fit Decreasing)");
    printf("Enter the number of items in Set: ");

    int n, i;
    scanf("%d", &n);
    printf("Enter %d items: ", n);
    int a[n];
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
    printf("Enter the bin size: ");
    int size;
    scanf("%d", &size);
    int *sequence = sort(a, n);
    binPacking(sequence, size, n);
    return 0;
}

```

Examples:



```

C:\Users\dbpat\Desktop\first_fit_decreasing.exe
BIN - PACKING Algorithm 1D Objects(First Fit Decreasing)Enter the number of items in Set: 5
Enter 5 items: 15 12 25 46 98
Enter the bin size: 60
Number of bins required using first fit decreasing algorithm is: 2
Process returned 0 (0x0)   execution time : 14.026 s
Press any key to continue.

```

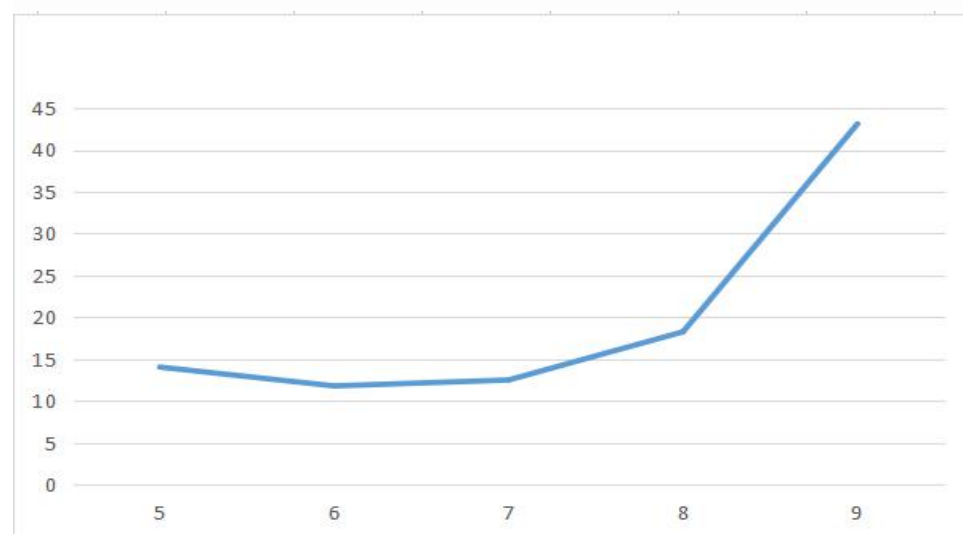
```
C:\Users\dbpat\Desktop\first_fit_decreasing.exe
BIN - PACKING Algorithm 1D Objects(First Fit Decreasing)Enter the number of items in Set: 6
Enter 6 items: 15 12 19 18 85 65
Enter the bin size: 60
Number of bins required using first fit decreasing algorithm is: 2
Process returned 0 (0x0)   execution time : 11.771 s
Press any key to continue.
```

```
C:\Users\dbpat\Desktop\first_fit_decreasing.exe
BIN - PACKING Algorithm 1D Objects(First Fit Decreasing)Enter the number of items in Set: 7
Enter 7 items: 15 25 65 95 85 7
45
Enter the bin size: 60
Number of bins required using first fit decreasing algorithm is: 2
Process returned 0 (0x0)   execution time : 12.485 s
Press any key to continue.
```

```
C:\Users\dbpat\Desktop\first_fit_decreasing.exe
BIN - PACKING Algorithm 1D Objects(First Fit Decreasing)Enter the number of items in Set: 8
Enter 8 items: 15
23 69 5 45 26 15 2
Enter the bin size: 60
Number of bins required using first fit decreasing algorithm is: 3
Process returned 0 (0x0)   execution time : 18.251 s
Press any key to continue.
```

```
C:\Users\dbpat\Desktop\first_fit_decreasing.exe
BIN - PACKING Algorithm 1D Objects(First Fit Decreasing)Enter the number of items in Set: 9
Enter 9 items: 1 76 65 32 64 34 89 23 59
Enter the bin size: 60
Number of bins required using first fit decreasing algorithm is: 3
Process returned 0 (0x0)   execution time : 43.115 s
Press any key to continue.
```

Graph:



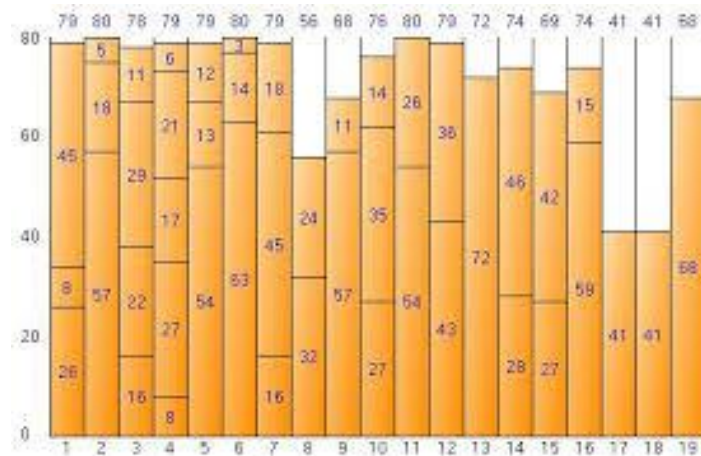
Offline Algorithm code:

Program:

A Graph Theoretic Approach for Minimizing Storage Space using Bin Packing Heuristics

In the age of Big Data the problem of storing huge volume of data in a minimum storage space by utilizing available resources properly is an open problem and an important research aspect in recent days. This problem has a close relationship with the famous classical NP-Hard combinatorial optimization problem namely the “Bin Packing Problem” where bins represent available storage space and the problem is to store the items or data in minimum number of bins. This research work mainly focuses on to find a near optimal solution of the offline one dimensional Bin Packing Problem based on two heuristics by taking the advantages of graph. Additionally, extreme computational results on some benchmark instances are reported and compared with the best known solution and solution produced by the four other well-known bin oriented heuristics. Also some future directions of the proposed work have been depicted.

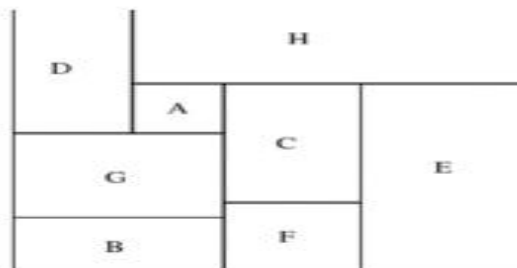
Some Graphs for Bin packing to understand properly:



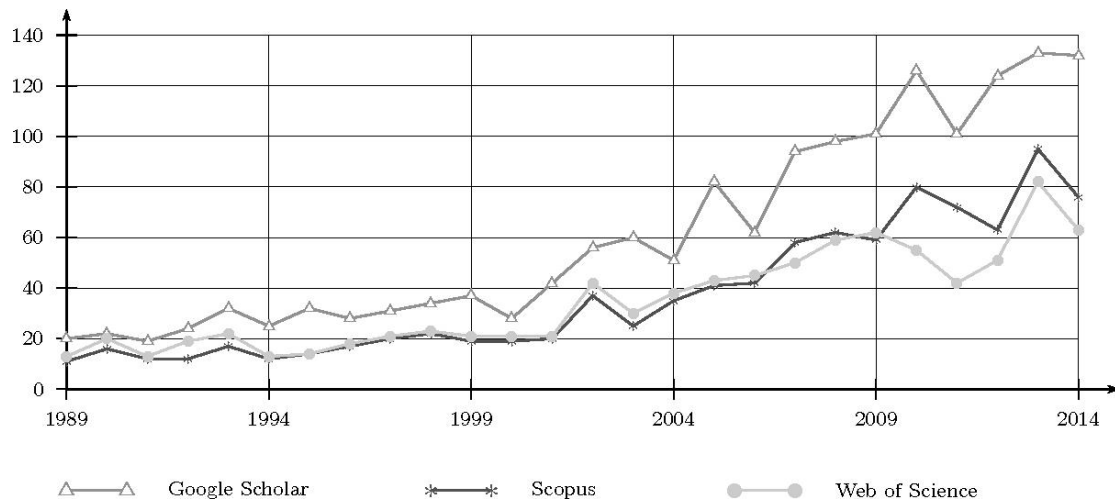
Classical Bin packing algorithm



Shows the memory allocation in Hard disk



A perfect arrangement of weights in a bin



A Memory optimisation ratio by bin packing by different companies in different years.

Conclusion:

-

Two approximation algorithms $A1$, and $A2$ were proposed in this paper. It was proved that the $A1$ approximation ratio is $3/2$. After that we observed the results of experimental simulations and analyzed them. Based on the results, we can claim that the two proposed algorithms in this article are the best presented approximation algorithms for the Bin Packing Problem, in theory and in practice until now.

In future researches, the focus on *Scaling Factor r* can enhance the algorithm $A2$ more and more.

So Heuristic approach is the better way for memory allocation in microprocessor which can also increase the speed of the computer and there will be revolution in searching things in element in the less amount of time .

Real Application Of Bin Packing

Here is a code for memory allocation in computer so that we can have a proper optimisation of memory so that the wastage of memory can be reduced and we can store more data. Research are going on for **Bin Packing Algorithm** so that we can store the data which is stored right now in 20th part of it.

Program:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void create(int,int);
void del(int);
void compaction();
void display();

int
fname[10],fsize[10],fstart[10],freest[10],freesize[10],m=0,n=0,start;

int main()
{
    int name,size,ch,i;
    int *ptr;
    ptr=(int *)malloc(sizeof(int)*100);
    start=freest[0]=(int)ptr;
    freesize[0]=500;

    printf("\n\n");
    printf(" Free start address          Free Size
\n\n");

    for(i=0;i<=m;i++)

printf("      %d                      %d\n",freest[i],freesize[i]);
    printf("\n\n");
    while(1)
    {
```

```

printf("1.Create.\n");
printf("2.Delete.\n");
printf("3.Compaction.\n");
printf("4.Exit.\n");
printf("Enter your choice: ");
scanf("%d",&ch);
switch(ch)
{
    case 1:
        printf("\nEnter the name of file: ");
        scanf("%d",&name);
        printf("\nEnter the size of the file: ");
        scanf("%d",&size);
        create(name,size);
        break;
    case 2:
        printf("\nEnter the file name which u want to delete:
");
        scanf("%d",&name);
        del(name);
        break;
    case 3:
        compaction();
        printf("\nAfter compaction the tables will be:\n");
        display();
        break;
    case 4:
        {
            exit(0);
        }
    default:
        printf("\nYou have entered a wrong choice.\n");
}
}
}

```

```

void create(int name,int size)
{
    int i,flag=1,j,a;

```

```

        for(i=0;i<=m;i++)
if( freesize[i] >= size)
    a=i,flag=0;
    if(!flag)
    {
        for(j=0;j<n;j++);
        n++;
        fname[j]=name;
        fsize[j]=size;
        fstart[j]=freest[a];
        freest[a]=freest[a]+size;
        freesize[a]=freesize[a]-size;

        printf("\n The memory map will now be:

\n\n");
        display();
    }
    else
    {
        printf("\nNo enough space is available.System
compaction.....");

        flag=1;

        compaction();
        display();

        for(i=0;i<=m;i++)
            if( freesize[i] >= size)
                a=i,flag=0;
        if(!flag)
        {
            for(j=0;j<n;j++);
            n++;
            fname[j]=name;
            fsize[j]=size;
            fstart[j]=freest[a];
            freest[a]+=size;
            freesize[a]-=size;
            printf("\n The memory map will now be: \n\n");

```

```

        display();
    }
    else
        printf("\nNo enough space.\n");
    }
}

void del(int name)
{
    int i,j,k,flag=1;
    for(i=0;i<n;i++)
        if(fname[i]==name)
            break;
    if(i==n)
    {
        flag=0;
        printf("\nNo such process exists.....\n");
    }
    else
    {
        m++;
        freest[m]=fstart[i];
        freesize[m]=fsize[i];
        for(k=i;k<n;k++)
        {
            fname[k]=fname[k+1];
            fsize[k]=fsize[k+1];
            fstart[k]=fstart[k+1];
        }
        n--;
    }
    if(flag)
    {
        printf("\n\n After deletion of this process the memory map
will be : \n\n");
        display();
    }
}

void compaction()
{

```

```

int i,j,size1=0,f_size=0;
    if(fstart[0]!=start)
    {
fstart[0]=start;
for(i=1;i<n;i++)
    fstart[i]=fstart[i-1]+fsize[i-1];
    }
else
    {
    for(i=1;i<n;i++)
        fstart[i]=fstart[i-1]+fsize[i-1];
    }
f_size=freesize[0];

for(j=0;j<=m;j++)
    size1+=freesize[j];
freest[0]=freest[0]-(size1-f_size);
freesize[0]=size1;
m=0;
}

void display()
{
    int i;

    printf("\n    ***    MEMORY MAP TABLE    ***\n");
    printf("\n\nNAME        SIZE        STARTING ADDRESS\n\n");
    for(i=0;i<n;i++)
        printf(" %d%10d%10d\n",fname[i],fsize[i],fstart[i]);
    printf("\n\n");
    printf("\n\n***    FREE SPACE TABLE    ***\n\n");
    printf("FREE START ADDRESS        FREE SIZE\n\n");
    for(i=0;i<=m;i++)

    printf("        %d        %d\n",freest[i],freesize[i]);
}

```

Output:

```
Select C:\Users\dbpat\Desktop\contiguous_memory_algorithm.exe

Free start address      Free Size
      45033728              500

1.Create.
2.Delete.
3.Compaction.
4.Exit.
Enter your choice: 1

Enter the name of file: 12

Enter the size of the file: 200

The memory map will now be:

***  MEMORY MAP TABLE  ***

NAME      SIZE      STARTING ADDRESS
12         200     45033728

***  FREE SPACE TABLE  ***

FREE START ADDRESS      FREE SIZE
      45033928              300

1.Create.
2.Delete.
3.Compaction.
4.Exit.
Enter your choice:
```

Thank you